A Formal Specification of the Cardano Consensus

Deliverable XXXX

Javier Díaz `<javier.diaz@iohk.io>`

Project: Cardano

Type: Deliverable
Due Date: XXXX

Responsible team: Formal Methods Team
Editor: Javier Díaz, IOHK
Team Leader: James Chapman, IOHK

Version 1.0
XXXX

Dissemination Level		
PU	Public	√
CO	Confidential, only for company distribution	
DR	Draft, not for general circulation	

Contents

List of Figures

Change Log

Rev.	Date	Who	Team	What
1	2024/06/20	Javier Díaz	FM (IOHK)	Initial version (0.1).

A Formal Specification of the Cardano Consensus

Javier Díaz
`javier.diaz@iohk.io`

October 3, 2024

Abstract

This document provides a formal specification of the Cardano consensus layer.

List of Contributors

...

1 Cryptographic Primitives

1.1 Serialization

– TODO: Add paragraph

Types & functions

```
Ser    : Type
encode : {T : Type} → T → Ser
decode : {T : Type} → Ser → Maybe T
_∥_    : Ser → Ser → Ser
```

Properties

```
∀ {T : Type} (x : T) → decode (encode x) ≡ just x
```

Figure 1: Definitions for serialization

1.2 Cryptographic Hashes

– TODO: Add paragraph and show only the relevant bits of the code below.

```
record isHashableSet (T : Type) : Set₁ where
  constructor mkIsHashableSet
  field THash : Type
        ⦃ DecEq-THash ⦄ : DecEq THash
        ⦃ Show-THash ⦄ : Show   THash
        ⦃ DecEq-T     ⦄ : DecEq T
        ⦃ T-Hashable ⦄ : Hashable T THash
open isHashableSet

record HashableSet : Type₁ where
  constructor mkHashableSet
  field T : Type; ⦃ T-isHashable ⦄ : isHashableSet T
  open isHashableSet T-isHashable public
```

1.3 Public-Key Cryptography

The Cardano blockchain system is based on a public-key cryptographic system.

Types & functions

```
SKey VKey : Type
isKeyPair : SKey → VKey → Type

KeyPair = Σ[ sk ∈ SKey ] Σ[ vk ∈ VKey ] isKeyPair sk vk
```

Figure 2: Definitions for the public-key cryptographic system

Types & functions

```
Sig  : Type
isSigned : VKey → Ser → Sig → Type
sign : SKey → Ser → Sig
```

Properties

```
∀ ((sk , vk , _) : KeyPair) (d : Ser) (σ : Sig) → sign sk d ≡ σ → isSigned vk d σ
```

Figure 3: Definitions for the digital signature scheme

1.4 Digital Signatures

– TODO: Add paragraph.

1.5 Key-Evolving Signatures

– TODO: Add paragraph.

Types & functions

```
Sig  : Type
isSigned : VKey → ℕ → Ser → Sig → Type
sign : (ℕ → SKey) → ℕ → Ser → Sig
```

Properties

```
∀ (n : ℕ) (sk : ℕ → SKey) ((skₙ , vk , _) : KeyPair) (d : Ser) (σ : Sig)
  → skₙ ≡ sk n → sign sk n d ≡ σ → isSigned vk n d σ
```

Figure 4: Definitions for key-evolving signatures

1.6 Verifiable Random Functions

– TODO: Add paragraph.

Types & functions

```
Seed Proof : Type
verify   : {T : Type} → VKey → Seed → Proof × T → Type
evaluate : {T : Type} → SKey → Seed → Proof × T
_XOR_    : Seed → Seed → Seed
```

Properties

```
∀ {T : Type} ((sk , vk , _) : KeyPair) (seed : Seed)
  → verify {T = T} vk seed (evaluate sk seed)
```

Figure 5: Definitions for verifiable random functions

2 Transition Rule Dependencies

Figure 6 shows all STS rules, the sub-rules they use and possible dependencies. Each node in the graph represents one rule, the top rule being CHAINHEAD. A straight arrow from one node to another one represents a sub-rule relationship.

An arrow with a dotted line from one node to another represents a dependency in the sense that the output of the target rule is an input to the source one, either as part of the source state, the environment or the signal. These dependencies are between sub-rules of a rule.

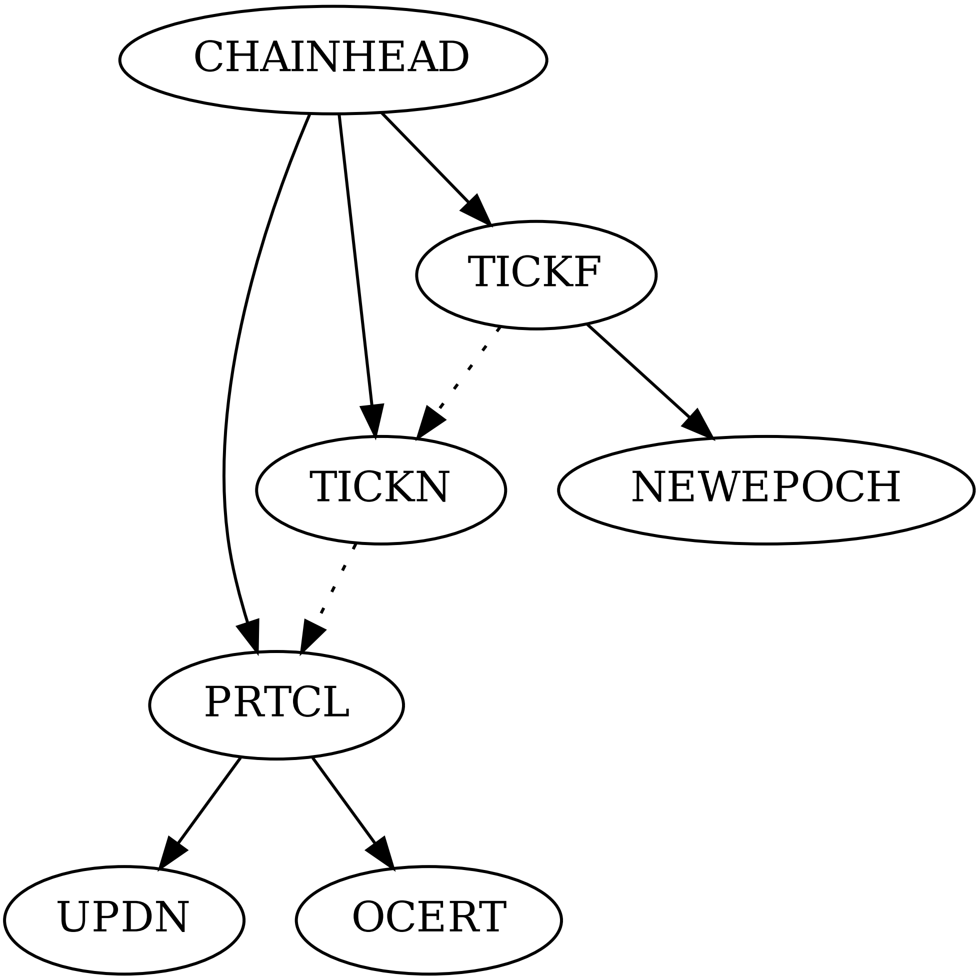

Figure 6: STS Rules, Sub-Rules and Dependencies

3 Ledger Interface

This section describes the interface exposed by the Ledger Layer which is used by the Consensus Layer.

```
NewEpochState       : Type
getPParams          : NewEpochState → PParams
getEpoch            : NewEpochState → Epoch
getPoolDistr        : NewEpochState → PoolDistr
adoptGenesisDelegs : NewEpochState → Slot → NewEpochState
_⊢_⇀⦇_,NEWEPOCH⦈_ : ⊤ → NewEpochState → Epoch → NewEpochState → Type
```

Figure 7: Ledger interface

4 Blockchain Layer

This chapter introduces the view of the blockchain layer as required for the consensus. The main transition rule is CHAINHEAD which calls the subrules TICKF, TICKN and PRTCL.

4.1 Block Definitions

Abstract types

```
HashHeader : Type -- hash of a block header
HashBBody : Type -- hash of a block body
VRFRes    : Type -- VRF result value
```

Concrete types

```
BlockNo = ℕ -- block number
Certifiedℕ = ∃[ n ] n < 2 ^ 512 -- [0, 2^512) (64-byte VRF output)
```

Operational Certificate

```
record OCert : Type where
  vkₕ : VKeyᵏ     -- operational (hot) key
  n  : ℕ         -- certificate issue number
  c₀ : KESPeriod -- start KES period
  σ  : Sigˢ       -- cold key signature
```

Block Header Body

```
record BHBody : Type where
  prevHeader : Maybe HashHeader -- hash of previous block header
  issuerVk : VKeyˢ               -- block issuer
  vrfVk    : VKeyᵛ               -- VRF verification key
  blockNo  : BlockNo             -- block number
  slot     : Slot                -- block slot
  vrfRes   : VRFRes              -- VRF result value
  vrfPrf   : Proof               -- VRF proof
  bodySize : ℕ                   -- size of the block body
  bodyHash : HashBBody           -- block body hash
  oc       : OCert               -- operational certificate
  pv       : ProtVer             -- protocol version
```

Block Types

```
BHeader = BHBody × Sigᵏ -- block header
```

Abstract functions

```
  headerHash     : BHeader → HashHeader -- hash of a block header
  headerSize     : BHeader → ℕ -- size of a block header
  slotToSeed     : Slot → Seed -- big-endian encoding of the slot number in 8 bytes
  prevHashToNonce : Maybe HashHeader → Nonce
  serHashToℕ     : SerHash → Certifiedℕ
  serHashToNonce : SerHash → Nonce
```

Figure 8: Block definitions

4.2 TICKN Transition

The Tick Nonce Transition (TICKN) is responsible for updating the epoch nonce and the previous epoch's hash nonce at the start of an epoch. Its environment is shown in Figure 9 and consists of the candidate nonce `ηc` and the previous epoch's last block header hash as a nonce `ηph`. Its state consists of the epoch nonce `η₀` and the previous epoch's last block header hash nonce `ηh`.

Tick Nonce environments

```
record TickNonceEnv : Type where
    ηc : Nonce  -- candidate nonce
    ηph : Nonce -- previous header hash as nonce
```

Tick Nonce states

```
record TickNonceState : Type where
    η₀ : Nonce -- epoch nonce
    ηh : Nonce -- nonce from hash of previous epoch's last block header
```

Tick Nonce transitions

```
  _⊢_⇀⦇_,TICKN⦈_ : TickNonceEnv → TickNonceState → Bool → TickNonceState → Type
```

Figure 9: Tick Nonce transition system types

The signal to the transition rule TICKN is a marker indicating whether we are in a new epoch. If we are in a new epoch, we update the epoch nonce and the previous hash. Otherwise, we do nothing. The TICKN rule is shown in Figure 10.

```
Not-New-Epoch :

  ────────────────────────────────
  ⟦ ηc , ηph ⟧ᵗᵉ ⊢ ⟦ η₀ , ηh ⟧ᵗˢ ⇀⦇ false ,TICKN⦈ ⟦ η₀ , ηh ⟧ᵗˢ

New-Epoch :

  ────────────────────────────────
  ⟦ ηc , ηph ⟧ᵗᵉ ⊢ ⟦ η₀ , ηh ⟧ᵗˢ ⇀⦇ true ,TICKN⦈ ⟦ ηc ⋆ ηh , ηph ⟧ᵗˢ
```

Figure 10: Tick Nonce transition system rules

4.3 UPDN Transition

The Update Nonce Transition (UPDN) updates the nonces until the randomness gets fixed. The environment is shown in Figure 11 and consists of the block nonce `η`. The state is shown in Figure 11 and consists of the candidate nonce `ηc` and the evolving nonce `ηv`.

Update Nonce environments

```
record UpdateNonceEnv : Type where
    η : Nonce -- new nonce
```

Update Nonce states

```
record UpdateNonceState : Type where
    ηv : Nonce -- evolving nonce
    ηc : Nonce -- candidate nonce
```

Update Nonce transitions

```
  _⊢_⇀⦇_,UPDN⦈_ : UpdateNonceEnv → UpdateNonceState → Slot → UpdateNonceState → Type
```

Figure 11: Update Nonce transition system types

The transition rule UPDN takes the slot s as signal and is shown in Figure 12. There are two different cases for UPDN: one where s is not yet `RandomnessStabilisationWindow`[1] slots from the beginning of the next epoch and one where s is less than `RandomnessStabilisationWindow` slots until the start of the next epoch.

Note that in the first rule, the candidate nonce ηc transitions to $\eta v \star \eta$, not $\eta c \star \eta$. The reason for this is that even though the candidate nonce is frozen sometime during the epoch, we want the two nonces to again be equal at the start of a new epoch.

```
Update-Both :
  • s + RandomnessStabilisationWindow < firstSlot (sucᵉ (epoch s))
  ────────────────────────────────
  ⟦ η ⟧ᵘᵉ ⊢ ⟦ ηv , ηc ⟧ᵘˢ ⇀⦇ s ,UPDN⦈ ⟦ ηv ⋆ η , ηv ⋆ η ⟧ᵘˢ

Only-Evolve :
  • s + RandomnessStabilisationWindow ≥ firstSlot (sucᵉ (epoch s))
  ────────────────────────────────
  ⟦ η ⟧ᵘᵉ ⊢ ⟦ ηv , ηc ⟧ᵘˢ ⇀⦇ s ,UPDN⦈ ⟦ ηv ⋆ η , ηc ⟧ᵘˢ
```

Figure 12: Update Nonce transition system rules

---

[1] Note that in pre-Conway eras `StabilityWindow` was used instead of `RandomnessStabilisationWindow`.

4.4 OCERT Transition

The Operational Certificate Transition (OCERT) validates the operational certificate and signature in the block header and updates the mapping of operational certificate issue numbers. The environment is shown in Figure 13 and consists of the set of stake pools. The state is shown in Figure 13 and consists of the mapping of operation certificate issue numbers. Its signal is a block header.

Operational Certificate environments

```
OCertEnv = ℙ KeyHashˢ
```

Operational Certificate states

```
OCertState = OCertCounters
```

Operational Certificate transitions

```
_⊢_⇀⦇_,OCERT⦈_ : OCertEnv → OCertState → BHeader → OCertState → Type
```

Operational Certificate helper function

```
currentIssueNo : OCertEnv → OCertState → KeyHashˢ → Maybe ℕ
currentIssueNo stpools cs hk =
  if hk ∈ dom (cs ˢ) then
    just (lookupᵐ cs hk)
  else
  if hk ∈ stpools then
    just 0
  else
    nothing
```

Figure 13: Operational Certificate transition-system types and functions

The transition rule OCERT is shown in Figure 14. From the block header body *bhb* we first extract the following:

- The operational certificate `oc`, consisting of the hot key vk_h, the certificate issue number n, the KES period start c_0 and the cold key signature τ.
- The cold key `issuerVk`.
- The slot `slot` for the block.
- The number t of KES periods that have elapsed since the start period on the certificate.

Using this we verify the preconditions of the operational certificate state transition which are the following:

- The KES period kp of the slot in the block header body must be greater than or equal to the start value c_0 listed in the operational certificate, and less than `MaxKESEvo`-many KES periods after c_0. The value of `MaxKESEvo` is the agreed-upon lifetime of an operational certificate, see [2].
- hk exists as key in the mapping of certificate issues numbers to a KES period m and that period is less than or equal to n. Also, n must be less than or equal to the successor of m.

- The signature τ can be verified with the cold verification key `issuerVk`.
- The KES signature σ can be verified with the hot verification key vk_h.

After this, the transition system updates the operational certificate state by updating the mapping of operational certificates where it overwrites the entry of the key hk with the KES period n.

```
Update-OCert :
  let (bhb , σ) = bh; open BHBody bhb
      ⟦ vkₕ , n , c₀ , τ ⟧ᵒᶜ = oc
      hk = hash issuerVk
      kp = kesPeriod slot
      t = kp -ᵏ c₀
  in
  • c₀ ≤ kp
  • kp < c₀ +ᵏ MaxKESEvo
  • ∃[ m ] (just m ≡ currentIssueNo stpools cs hk × (n ≡ m ⊎ n ≡ suc m))
  • isSignedˢ issuerVk (encode (vkₕ , n , c₀)) τ
  • isSignedᵏ vkₕ t (encode bhb) σ
  ────────────────────────────────
  stpools ⊢ cs —⇀⦇ bh ,OCERT⦈ (❴ hk , n ❵ ∪ˡ cs)
```

Figure 14: Operational Certificate transition-system rules

The OCERT rule has the following predicate failures:

1. If the KES period is less than the KES period start in the certificate, there is a KESBeforeStart failure.
2. If the KES period is greater than or equal to the KES period end (start + `MaxKESEvo`) in the certificate, there is a KESAfterEnd failure.
3. If the period counter in the original key hash counter mapping is larger than the period number in the certificate, there is a CounterTooSmall failure.
4. If the period number in the certificate is larger than the successor of the period counter in the original key hash counter mapping, there is a CounterOverIncremented failure.
5. If the signature of the hot key, KES period number and period start is incorrect, there is an InvalidSignature failure.
6. If the KES signature using the hot key of the block header body is incorrect, there is an InvalideKesSignature failure.
7. If there is no entry in the key hash to counter mapping for the cold key, there is a NoCounterForKeyHash failure.

4.5 PRTCL Transition

The Protocol Transition (PRTCL) calls the transition UPDN to update the evolving and candidate nonces, and checks the operational certificate with OCERT. Its environment is shown in Figure 15 and consists of:

- The stake pool stake distribution `pd`.
- The epoch nonce `η₀`.

Its state is shown in Figure 15 and consists of

- The operational certificate issue number mapping `cs`.
- The evolving nonce `ηv`.
- The candidate nonce for the next epoch `ηc`.

Protocol environments

```
record PrtclEnv : Type where
  pd : PoolDistr -- pool stake distribution
  η₀ : Nonce -- epoch nonce
```

Protocol states

```
record PrtclState : Type where
  cs : OCertCounters -- operational certificate issues numbers
  ηv : Nonce -- evolving nonce
  ηc : Nonce -- candidate nonce
```

Protocol transitions

```
 _⊢_⇀⦇_,PRTCL⦈_ : PrtclEnv → PrtclState → BHeader → PrtclState → Type
```

Figure 15: Protocol transition system types

In Figure 16 we define a function `vrfChecks` which performs all the VRF related checks on a given block header body. In addition to the block header body *bhb*, the function requires the epoch nonce η_0, the stake distribution *pd* (aggregated by pool), and the active slots coefficient f from the protocol parameters. The function checks:

- The validity of the proofs `vrfPrf` for the leader value and the new nonce.
- The verification key *vrfHK* is associated with relative stake σ in the stake distribution.
- The `hBLeader` value of *bhb* indicates a possible leader for this slot. The function `checkLeaderVal`, defined in Figure 16, performs this check.

The function `vrfChecks` has the following predicate failures:

1. If the VRF key is not in the pool distribution, there is a VRFKeyUnknown failure.
2. If the VRF key hash does not match the one listed in the block header, there is a VRFKeyWrongVRFKey failure.

Protocol helper functions

```
hBLeader : BHBody → Certifiedℕ
hBLeader bhb = serHashToℕ (hash (encode "L" ∥ encode vrfRes))
  where open BHBody bhb

hBNonce : BHBody → Nonce
hBNonce bhb = serHashToNonce (hash (encode "N" ∥ encode vrfRes))
  where open BHBody bhb

checkLeaderVal : Certifiedℕ → PosUnitInterval → ℚ → Type
checkLeaderVal (certℕ , certℕprf) (f , posf , f≤1) σ =
  if f ≡ 1ℚ then ⊤ else
  let
    p = pos certℕ ℚ./ (2 ^ 512)
    q = 1ℚ ℚ.- p
    c = ln (1ℚ ℚ.- f)
  in
    ℚ.1/ q ℚ.< exp ((ℚ.- σ) ℚ.* c)

vrfChecks : Nonce → PoolDistr → PosUnitInterval → BHBody → Type
vrfChecks η₀ pd f bhb =
  case lookupPoolDistr pd hk of
    λ where
      nothing → ⊥
      (just (σ , vrfHK)) →
        vrfHK ≡ hash vrfVk
        × verify vrfVk seed (vrfPrf , vrfRes)
        × checkLeaderVal (hBLeader bhb) f σ
  where
    open BHBody bhb
    hk = hash issuerVk
    seed = slotToSeed slot XOR nonceToSeed η₀
```

Figure 16: Protocol transition system helper functions

3. If the VRF generated value in the block header does not validate against the VRF certificate, there is a VRFKeyBadProof failure.
4. If the VRF generated leader value in the block header is too large compared to the relative stake of the pool, there is a VRFLeaderValueTooBig failure.

The transition rule PRTCL is shown in Figure 17.

```
Evolve-Prtcl :
  let (bhb , σ) = bh; open BHBody bhb
      η = hBNonce bhb
  in
  • ⟦ η ⟧ᵘᵉ ⊢ ⟦ ηv , ηc ⟧ᵘˢ ⇀⦇ slot ,UPDN⦈ ⟦ ηv′ , ηc′ ⟧ᵘˢ
  • dom (pd ˢ) ⊢ cs ⇀⦇ bh ,OCERT⦈ cs′
  • vrfChecks η₀ pd ActiveSlotCoeff bhb
  ────────────────────────────────
  ⟦ pd , η₀ ⟧ᵖᵉ ⊢ ⟦ cs , ηv , ηc ⟧ᵖˢ ⇀⦇ bh ,PRTCL⦈ ⟦ cs′ , ηv′ , ηc′ ⟧ᵖˢ
```

Figure 17: Protocol transition system rules

4.6 TICKF Transition

The Tick Forecast Transition (TICKF) performs some chain level upkeep. Its state is shown in Figure 18 and consists of the epoch specific state `NewEpochState` necessary for the NEWEPOCH transition and its signal is the current slot.

Tick Forecast transitions

```
_⊢_⇀⦇_,TICKF⦈_ : ⊤ → NewEpochState → Slot → NewEpochState → Type
```

Figure 18: Tick forecast transition system types

The transition TICKF is shown in Figure 19. Part of the upkeep is updating the genesis key delegation mapping according to the future delegation mapping using the helper function `adoptGenesisDelegs`. One sub-transition is done: The NEWEPOCH transition performs any state change needed if it is the first block of a new epoch.

```
Tick-Forecast :
  let forecast = adoptGenesisDelegs nes' s
  in
  • _ ⊢ nes ⇀⦇ epoch s ,NEWEPOCH⦈ nes'
  ────────────────────────────────────
  _ ⊢ nes ⇀⦇ s ,TICKF⦈ forecast
```

Figure 19: Tick forecast transition system rules

4.7 CHAINHEAD Transition

The Chain Head Transition rule (CHAINHEAD) is the main rule of the blockchain layer part of the STS. It calls TICKF, TICKN, and PRTCL, as sub-rules.

Its state is shown in Figure 20 and consists of the epoch specific state `NewEpochState` and its signal is a block header. Its state is shown in Figure 20 and it consists of the following:

- The operational certificate issue number map `cs`.
- The epoch nonce η_0.
- The evolving nonce `ηv`.
- The candidate nonce `ηc`.
- The previous epoch hash nonce `ηh`.
- The last header hash `h`.
- The last slot `sℓ`.
- The last block number `bℓ`.

The transition checks the following things (via the functions `chainChecks` and `prtlSeqChecks` from Figure 20):

- The slot in the block header body is larger than the last slot recorded.
- The block number increases by exactly one.
- The previous hash listed in the block header matches the previous block header hash which was recorded.
- The size of the block header is less than or equal to the maximal size that the protocol parameters allow for block headers.
- The size of the block body, as claimed by the block header, is less than or equal to the maximal size that the protocol parameters allow for block bodies.
- The node is not obsolete, meaning that the major component of the protocol version in the protocol parameters is not bigger than the constant `MaxMajorPV`.

The transition rule CHAINHEAD is shown in Figure 21 and has the following predicate failures:

1. If the slot of the block header body is not larger than the last slot, there is a WrongSlotInterval failure.
2. If the block number does not increase by exactly one, there is a WrongBlockNo failure.
3. If the hash of the previous header of the block header body is not equal to the last header hash, there is a WrongBlockSequence failure.
4. If the size of the block header is larger than the maximally allowed size, there is a HeaderSizeTooLarge failure.
5. If the size of the block body is larger than the maximally allowed size, there is a BlockSizeTooLarge failure.
6. If the major component of the protocol version is larger than `MaxMajorPV`, there is a ObsoleteNode failure.

Chain Head environments

```
ChainHeadEnv = NewEpochState
```

Chain Head states

```
record LastAppliedBlock : Type where
    bℓ : BlockNo -- last block number
    sℓ : Slot    -- last slot
    h : HashHeader -- latest header hash

record ChainHeadState : Type where
    cs : OCertCounters -- operational certificate issue numbers
    η₀ : Nonce         -- epoch nonce
    ηv : Nonce         -- evolving nonce
    ηc : Nonce         -- candidate nonce
    ηh : Nonce         -- nonce from hash of last epoch's last header
    lab : Maybe LastAppliedBlock -- latest applied block
```

Chain Head transitions

```
    _⊢_⇀⦇_,CHAINHEAD⦈_ : ChainHeadEnv → ChainHeadState → BHeader → ChainHeadState → Type
```

Chain Head helper functions

```
chainChecks : ℕ → ℕ × ℕ × ProtVer → BHeader → Type
chainChecks maxpv (maxBHSize , maxBBSize , protocolVersion) bh =
  m ≤ maxpv × headerSize bh ≤ maxBHSize × bodySize ≤ maxBBSize
  where
    m = proj₁ protocolVersion
    open BHBody (proj₁ bh)

lastAppliedHash : Maybe LastAppliedBlock → Maybe HashHeader
lastAppliedHash nothing = nothing
lastAppliedHash (just ⟦ _ , _ , h ⟧ℓ) = just h

prtlSeqChecks : Maybe LastAppliedBlock → BHeader → Type
prtlSeqChecks nothing bh = ⊤
prtlSeqChecks lab@(just ⟦ bℓ , sℓ , _ ⟧ℓ) bh = sℓ < slot × bℓ + 1 ≡ blockNo × ph ≡ prevHeader
  where
    open BHBody (proj₁ bh)
    ph = lastAppliedHash lab
```

Figure 20: Chain Head transition system types and functions

```
Chain-Head :
  let (bhb , _) = bh; open BHBody bhb
      e₁  = getEpoch nes
      e₂  = getEpoch forecast
      ne  = (e₁ ≢ e₂)
      pp  = getPParams forecast; open PParams
      nₚₕ = prevHashToNonce (lastAppliedHash lab)
      pd  = getPoolDistr forecast
      lab′ = just ⟦ blockNo , slot , headerHash bh ⟧ℓ
  in
  • prtlSeqChecks lab bh
  • _ ⊢ nes ⇀⦇ slot ,TICKF⦈ forecast
  • chainChecks MaxMajorPV (pp .maxHeaderSize , pp .maxBlockSize , pp .pv) bh
  • ⟦ ηc , nₚₕ ⟧ᵗᵉ ⊢ ⟦ η₀ , ηh ⟧ᵗˢ ⇀⦇ ne ,TICKN⦈ ⟦ η₀′ , ηh′ ⟧ᵗˢ
  • ⟦ pd , η₀′ ⟧ᵖᵉ ⊢ ⟦ cs , ηv , ηc ⟧ᵖˢ ⇀⦇ bh ,PRTCL⦈ ⟦ cs′ , ηv′ , ηc′ ⟧ᵖˢ
  ────────────────────────────────
  nes ⊢ ⟦ cs , η₀ , ηv , ηc , ηh , lab ⟧ᶜˢ ⇀⦇ bh ,CHAINHEAD⦈
        ⟦ cs′ , η₀′ , ηv′ , ηc′ , ηh′ , lab′ ⟧ᶜˢ
```

Figure 21: Chain Head transition system rules

5 Properties

This section describes the properties that the consensus layer should have. The goal is to include these properties in the executable specification to enable e.g. property-based testing or formal verification.

5.1 Header-Only Validation

In any given chain state, the consensus layer needs to be able to validate the block headers without having to download the block bodies. Property 5.1 states that if an extension of a chain that spans less than $\mathsf{StabilityWindow}$ slots is valid, then validating the headers of that extension is also valid. This property is useful for its converse: if the header validation check for a sequence of headers does not pass, then we know that the block validation that corresponds to those headers will not pass either. In these properties, we refer to the CHAIN transition system as defined in [3].

Property 5.1 (Header only validation). For all states s with slot number t[2], and chain extensions E with corresponding headers H such that:

$$0 \leq t_E - t \leq \mathsf{StabilityWindow}$$

we have:

$$\vdash s \xrightarrow[\text{chain}]{E}{}^{*} s' \implies nes \vdash \tilde{s} \xrightarrow[\text{chainhead}]{H}{}^{*} \tilde{s}'$$

where $s = (nes,\ \tilde{s})$, t_E is the maximum slot number appearing in the blocks contained in E, and H is obtained from E by extracting the header from each block in E.

Property 5.2 (Body only validation). For all states s with slot number t, and chain extensions $E = [b_0, \ldots, b_n]$ with corresponding headers $H = [h_0, \ldots, h_n]$ such that:

$$0 \leq t_E - t \leq \mathsf{StabilityWindow}$$

we have that for all $i \in [1, n]$:

$$nes \vdash \tilde{s} \xrightarrow[\text{chainhead}]{H}{}^{*} s_h \wedge \vdash (nes,\ \tilde{s}) \xrightarrow[\text{chain}]{[b_0,\ldots,b_{i-1}]}{}^{*} s_{i-1} \implies nes' \vdash \tilde{s}_{i-1} \xrightarrow[\text{chainhead}]{h_i} s'_h$$

where $s = (nes,\ \tilde{s})$, $s_{i-1} = (nes',\ \tilde{s}_{i-1})$, t_E is the maximum slot number appearing in the blocks contained in E.

Property 5.2 states that if we validate a sequence of headers, we can validate their bodies independently and be sure that the blocks will pass the chain validation rule. To see this, given an environment e and initial state s, assume that a sequence of headers $H = [h_0, \ldots, h_n]$ corresponding to blocks in $E = [b_0, \ldots, b_n]$ is valid according to the $\mathsf{CHAINHEAD}$ transition system:

$$nes \vdash \tilde{s} \xrightarrow[\text{chainhead}]{H}{}^{*} \tilde{s}'$$

Assume the bodies of E are valid according to the BBODY rules (defined in [3]), but E is not valid according to the CHAIN rule. Assume that there is a $b_j \in E$ such that it is the first block such that does not pass the CHAIN validation. Then:

$$\vdash (nes,\ \tilde{s}) \xrightarrow[\text{chain}]{[b_0,\ldots,b_{j-1}]}{}^{*} s_j$$

---

[2] i.e. the component s_ℓ of the last applied block of s equals t

But by Property 5.2 we know that

$$\mathit{nes}_j \vdash \tilde{s}_j \xrightarrow[\text{chainhead}]{h_j} \tilde{s}_{j+1}$$

which means that block b_j has valid headers, and this in turn means that the validation of b_j according to the chain rules must have failed because it contained an invalid block body. But this contradicts our assumption that the block bodies were valid.

Values associated with the leader value calculations

$\mathit{certNat} \in \{n \mid n \in \mathbb{N}, n \in [0, 2^{512})\}$	Certified natural value from VRF
$f \in [0,1]$	Active slot coefficient
$\sigma \in [0,1]$	Stake proportion

6 Leader Value Calculation

This section details how we determine whether a node is entitled to lead (under the Praos protocol) given the output of its verifiable random function calculation.

6.1 Computing the leader value

The verifiable random function gives us a 64-byte random output. We interpret this as a natural number *certNat* in the range $[0, 2^{512})$.

6.2 Node eligibility

As per [1], a node is eligible to lead when its leader value $p < 1 - (1-f)^{\sigma}$. We have

$$\begin{aligned} & p < 1 - (1-f)^{\sigma} \\ \iff & \left(\frac{1}{1-p}\right) < \exp\left(-\sigma \cdot \ln\left(1-f\right)\right) \end{aligned}$$

The latter inequality can be efficiently computed through use of its Taylor expansion and error estimation to stop computing terms once we are certain that the result will be either above or below the target value.

We carry out all computations using fixed precision arithmetic (specifically, we use 34 decimal bits of precision, since this is enough to represent the fraction of a single lovelace.)

As such, we define the following:

$$\begin{aligned} p &= \frac{\mathit{certNat}}{2^{512}} \\ q &= 1 - p \\ c &= \ln\left(1-f\right) \end{aligned}$$

and define the function checkLeaderVal as follows:

$$\mathsf{checkLeaderVal}\ \mathit{certNat}\ \sigma\ f = \begin{cases} \mathsf{True}, & f = 1 \\ \frac{1}{q} < \exp\left(-\sigma \cdot c\right), & \text{otherwise} \end{cases}$$

References

[1] Bernardo David, Peter Gaži, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake protocol. Cryptology ePrint Archive, Paper 2017/573, 2017. https://eprint.iacr.org/2017/573.

[2] IOHK Formal Methods Team. Design specification for delegation and incentives in cardano, iohk deliverable sl-d1. https://github.com/intersectmbo/cardano-ledger/releases/latest/download/shelley-delegation.pdf, 2018.

[3] IOHK Formal Methods Team. A formal specification of the cardano ledger. https://github.com/intersectmbo/cardano-ledger/releases/latest/download/shelley-ledger.pdf, 2019.